第 12 章　簡單工廠（Simple Factory）模式

簡單工廠模式是類別的建構模式，又叫做靜態工廠方法（Static Factory Method）模式。簡單工廠模式是由一個工廠物件決定建構出那一種產品類別的實例。

12.1　工廠模式的幾種形態

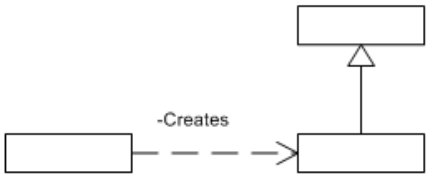
工廠模式專門負責將大量有共同介面的類別實例化。工廠模式可以動態決定將哪一個類別實例化，不必事先知道每次要實例化哪一個類別。工廠模式有以下幾種形態：

（1）簡單工廠（Simple Factory）模式，又稱靜態工廠方法模式（Static Factory Method Pattern）。

（2）工廠方法（Factory Method）模式，又稱多型工廠（Polymorphic Factory）模式或虛擬建構方法（Virtual Constructor）模式；

（3）抽象工廠（Abstract Factory）模式，又稱工具箱（Kit 或 Toolkit）模式。

下面就是簡單工廠模式的簡略類別圖。



簡單工廠模式，或稱靜態工廠方法模式，是不同的工廠方法模式的一個特殊實作。在其他文獻中，簡單工廠往往作爲普通工廠模式的一個特例討論。

在 Java 語言中，通常的工廠方法模式不能通過設計功能的退化給出靜態工廠方法模式。因爲一個方法是不是靜態的，對於 Java 語言來說是一個很大的區別，必須在一開始的時候就加以考慮。這就是本書將簡單工廠單獨提出來討論的一個原因。學習簡單工廠模式是對學習工廠方法模式的一個很好的準備，也是對學習其他模式，特別是單例模式和多例模式的一個很好的準備，這就是本書首先講解這一模式的另一個原因。

12.2 簡單工廠模式的引進

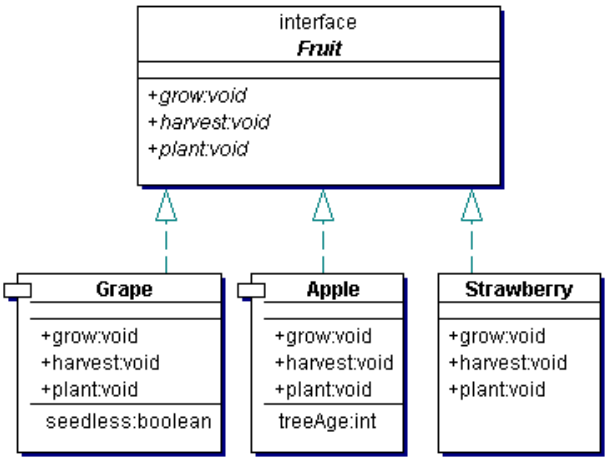
比如說有一個農場公司，專門向市場銷售各類水果。在這個系統裏需要描述下列的水果：

葡萄 Grape

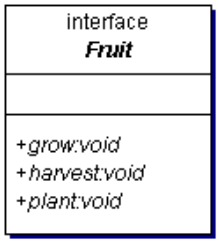
草莓 Strawberry

蘋果 Apple

水果與其他的植物有很大的不同，就是水果最終是可以採摘食用的。那麼一個自然的作法就是建立一個各種水果都適用的介面，以便與農場裏的其他植物區分開。如下圖所示。



水果介面規定出所有的水果必須實作的介面，包括任何水果類別必須具備的方法：種植 plant()，生長 grow()以及收穫 harvest()。介面 Fruit 的類別圖如下所示。



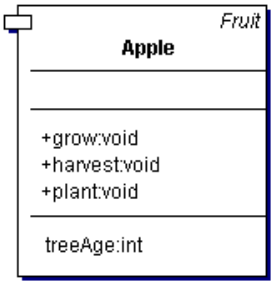
這個水果介面的程式碼如下所示。

程式碼清單 1：介面 Fruit 的程式碼

```
public interface Fruit
{
    /**
     *  生長
     */
```

```
    void grow();
  /**
   *  收穫
   */
    void harvest();
  /**
   *  種植
   */
    void plant();
}
```

描述蘋果的 Apple 類別的程式碼的類別圖如下所示。



Apple 類別是水果類別的一種，因此它實作了水果介面所宣告的所有方法。另外，由於蘋果是多年生植物，因此多出一個 treeAge 性質，描述蘋果樹的樹齡。下面是這個蘋果類別的程式碼。

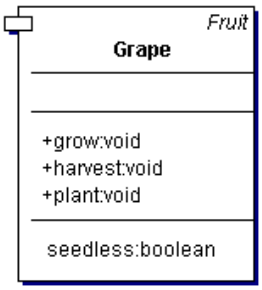
程式碼清單 2：類別 Apple 的程式碼

```
public class Apple implements Fruit
{
    private int treeAge;

  /**
   *  生長
   */
    public void grow()
    {
        log("Apple is growing...");
    }
  /**
   *  收穫
   */
    public void harvest()
    {
        log("Apple has been harvested.");
    }
  /**
```

```
    *  種植
    */
    public void plant()
    {
        log("Apple has been planted.");
    }
    /**
    *  輔助方法
    */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
    /**
    *  樹齡的取值方法
    */
    public int getTreeAge()
    {
        return treeAge;
    }
    /**
    *  樹齡的賦值方法
    */
    public void setTreeAge(int treeAge)
    {
        this.treeAge = treeAge;
    }
}
```

同樣，Grape 類別是水果類別的一種，也實作了 Fruit 介面所宣告的所有的方法。但由於葡萄分有籽和無籽兩種，因此，比通常的水果多出一個 seedless 性質，如下圖所示。



葡萄類別的程式碼如下所示。可以看出，Grape 類別同樣實作了水果介面，從而是水果類型的一種子類型。

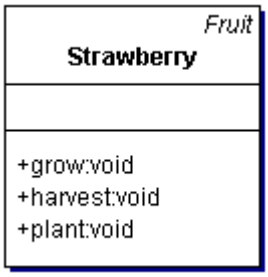
程式碼清單 3：類別 Grape 的程式碼

```
public class Grape implements Fruit
```

```
{
    private boolean seedless;

    /**
     *  生長
     */
    public void grow()
    {
        log("Grape is growing...");
    }
    /**
     *  收穫
     */
    public void harvest()
    {
        log("Grape has been harvested.");
    }
    /**
     *  種植
     */
    public void plant()
    {
        log("Grape has been planted.");
    }
    /**
     *  輔助方法
     */
    public static void log(String msg)
    {
        System.out.println(msg);
    }
    /**
     *  有無籽的取值方法
     */
    public boolean getSeedless()
    {
        return seedless;
    }
    /**
     *  有無籽的賦值方法
     */
    public void setSeedless(boolean seedless)
    {
        this.seedless = seedless;
    }
}
```

下圖所示是 Strawberry 類別的類別圖。

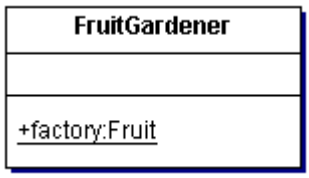


Strawberry 類別實作了 Fruit 介面，因此，也是水果類型的子類型，其程式碼如下所示。

程式碼清單 4：類別 Strawberry 的程式碼

```
public class Strawberry implements Fruit
{
    /**
     *  生長
     */
     public void grow()
     {
          log("Strawberry is growing...");
     }
    /**
     *  收穫
     */
     public void harvest()
     {
          log("Strawberry has been harvested.");
     }
    /**
     *  種植
     */
     public void plant()
     {
          log("Strawberry has been planted.");
     }
    /**
     *  輔助方法
     */
     public static void log(String msg)
     {
          System.out.println(msg);
     }
}
```

農場的園丁也是系統的一部分，自然要由一個合適的類別來代表。這個類別就是 FruitGardener 類別，其結構由下面的類別圖描述。



FruitGardener 類別會根據用戶端的要求，建構出不同的水果物件，比如蘋果（Apple），葡萄（Grape）或草莓（Strawberry）的實例。而如果接到不合法的要求，FruitGardener 類別會拋出 BadFruitException 例外，如下圖所示。

園丁類別的程式碼如下所示。

程式碼清單 5：FruitGardener 類別的程式碼

```
public class FruitGardener
{
    /**
     *  靜態工廠方法
     */
     public static Fruit factory(String which)
            throws BadFruitException
     {
          if (which.equalsIgnoreCase("apple"))
          {
               return new Apple();
          }
          else if (which.equalsIgnoreCase("strawberry"))
          {
               return new Strawberry();
          }
          else if (which.equalsIgnoreCase("grape"))
          {
               return new Grape();
          }
          else
          {
```

```
                throw new BadFruitException("Bad fruit request");
            }
        }
    }
```

可以看出，園丁類別提供了一個靜態工廠方法。在用戶端的呼叫下，這個方法建構用戶端所需要的水果物件。如果用戶端的請求是系統所不支援的，工廠方法就會拋出一個 BadFruitException 例外。這個例外類別的程式碼如下所示。

程式碼清單 6：BadFruitException 類別的程式碼

```
public class BadFruitException extends Exception
{
    public BadFruitException(String msg)
    {
        super(msg);
    }
}
```

在使用時，用戶端只需呼叫 FruitGardener 的靜態方法 factory()即可。請見下面的示意性用戶端程式碼。

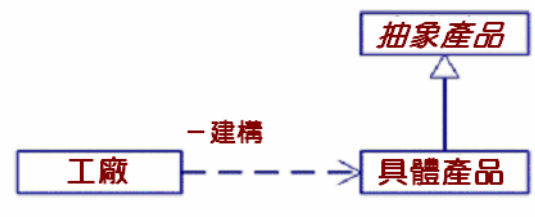
程式碼清單 7：怎樣使用例外類別 BadFruitException

```
try
{
    FruitGardener.factory("grape");
    FruitGardener.factory("apple");
    FruitGardener.factory("strawberry");
        ...
}
catch(BadFruitException e)
{
    ...
}
```

這樣，農場一定會百果豐收啦!

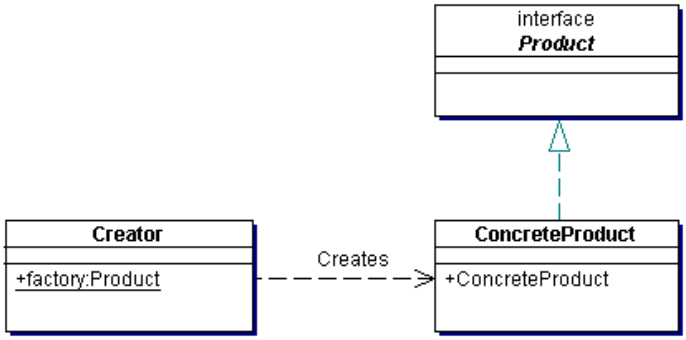
12.3 簡單工廠模式的結構

簡單工廠模式是類別的建構模式，這個模式的一般性結構如下圖所示。



角色與結構

簡單工廠模式就是由一個工廠類別可以根據傳入的參量決定建構出哪一種產品類別的實例。下圖所示爲以一個示意性的實作爲例說明簡單工廠模式的結構。



從上圖可以看出，簡單工廠模式涉及到工廠角色、抽象產品角色以及具體產品角色等三個角色：

- 工廠類別（Creator）角色：擔任這個角色的是工廠方法模式的核心，含有與應用緊密相關的商業邏輯。工廠類別在用戶端的直接呼叫下建構產品物件，它往往由一個具體 Java 類別實作。
- 抽象產品（Product）角色：擔任這個角色的類別是工廠方法模式所建構的物件的父類別，或它們共同擁有的介面。抽象產品角色可以用一個 Java 介面或者 Java 抽象類別實作。
- 具體產品（Concrete Product）角色：工廠方法模式所建構的任何物件都是這個角色的實例，具體產品角色由一個具體 Java 類別實作。

程式碼

工廠類別的示意性程式碼如下所示。可以看出，這個工廠方法建構了一個新的具體產品的實例並傳回給呼叫者。

程式碼清單 8：Creator 類別的程式碼

```
public class Creator
{
```

```
    /**
     *   靜態工廠方法
     */
     public static Product factory()
     {
          return new ConcreteProduct();
     }
}
```

抽象產品角色的主要目的是給所有的具體產品類別提供一個共同的類型，在最簡單的情況下，可以簡化爲一個標識介面。所謂標識介面，就是沒有宣告任何方法的空介面。關於標識介面的討論，請參見本書的“專題：Java 介面”一章。

程式碼清單 9：抽象角色 Product 介面的程式碼

```
public interface Product
{
}
```

具體產品類別的示意性程式碼如下。

程式碼清單 10：具體產品角色 ConcreteProduct 類別的程式碼

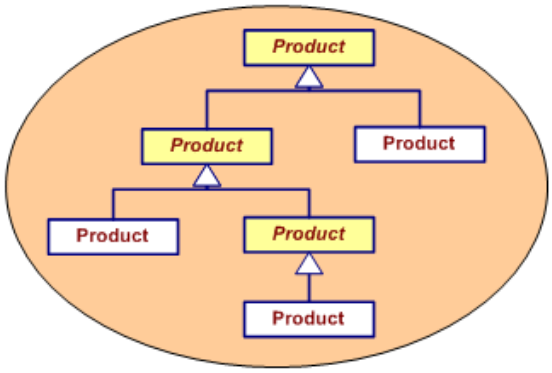
```
public class ConcreteProduct implements Product
{
    public ConcreteProduct(){}
}
```

雖然在這個簡單的示意性實作裏面只給出了一個具體產品類別，但是在實際應用中一般都會遇到多個具體產品類別的情況。

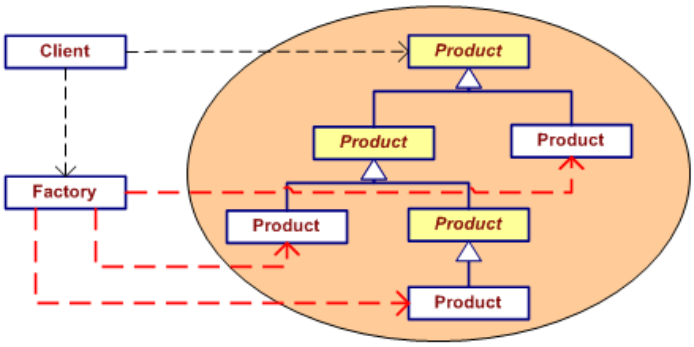
12.4 簡單工廠模式的實作

多層次的產品結構

在真實的系統中，產品可以形成複雜的層級結構，比如下圖所示的樹狀結構上就有多個抽象產品類別和具體產品類別。



這個時候，簡單工廠模式採取的是以不變應萬變的策略，一律使用同一個工廠類別。如下圖所示。

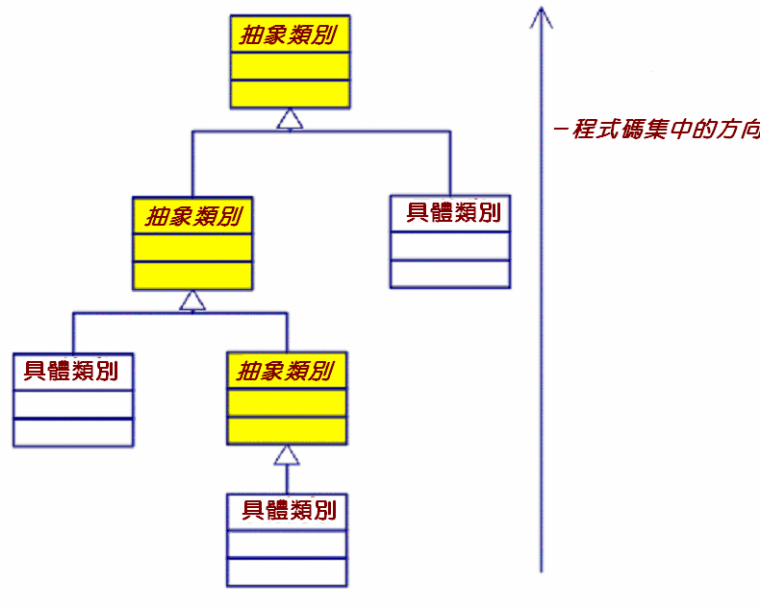


圖中從 Factory 類別到各個 Product 類別的虛線代表建構（依賴）關係；從 Client 到其他類別的聯線是一般依賴關係。

這樣做的好處是設計簡單，產品類別的層級結構不會反映到工廠類別中來，從而產品類別的層級結構的變化也就不會影響到工廠類別。但是這樣做的缺點是，增加新的產品必將導致工廠類別的修改。請參見後面“模式的優點和缺點”一節，以及本書的“工廠方法（Factory Method）模式”一章。

使用 Java 介面或者 Java 抽象類別

如果模式所產生的具體產品類別彼此之間沒有共同的商業邏輯，那麼抽象產品角色可以由一個 Java 介面扮演；相反，如果這些具體產品類別彼此之間確有共同的商業邏輯，那麼這些公有的邏輯就應當移到抽象角色裏面，這就意味著抽象角色應當由一個抽象類別扮演。在一個類型的層級結構裏面，共同的程式碼應當儘量向上移動，以達到共用的目的，如下圖所示。



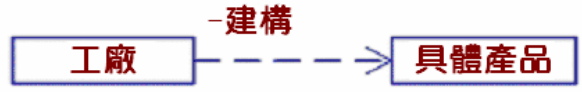
關於抽象類別與 Java 介面的關係與區別，請參見本書的“專題：Java 介面”與“專題：抽象類別”兩章。

多個工廠方法

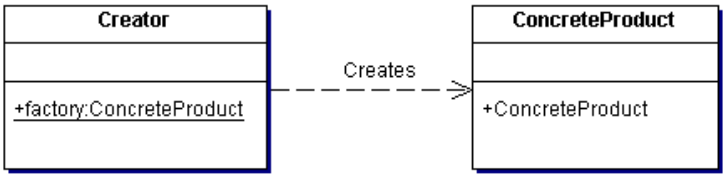
每個工廠類別可以有多於一個的工廠方法，分別負責建構不同的產品物件。比如 java.text.DateFormat 類別是其子類別的工廠類別，而 DateFormat 類別就提供了多個靜態工廠方法。請參見本章後面的詳盡講解。

抽象產品角色的省略

如果系統僅有一個具體產品角色的話，那麼就可以省略掉抽象產品角色。省略掉抽象產品類別後的簡略類別圖如下圖所示。



仍然以前面給出的示意性系統爲例，這時候系統的類別圖就變成如下所示。



下面是工廠類別的程式碼。顯然，這個類別提供一個工廠方法，傳回一個具體產品類別的實例。

程式碼清單 11：工廠角色的程式碼

```
package com.javapatterns.simplefactory.simplified;
public class Creator
{
    /**
     *  靜態工廠方法
     */
     public static ConcreteProduct factory()
     {
          return new ConcreteProduct();
     }
}
```

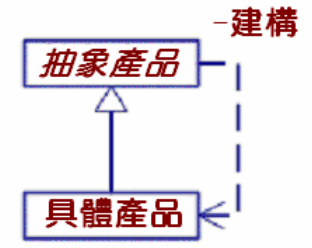
具體產品類別的程式碼如下所示。

程式碼清單 12：具體產品角色 ConcreteProduct 類別的程式碼

```
package com.javapatterns.simplefactory.simplified;
public class ConcreteProduct
{
     public ConcreteProduct(){}
}
```

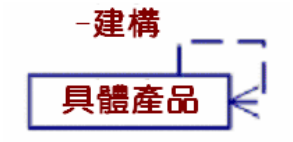
工廠角色與抽象產品角色合併

在有些情況下，工廠角色可以由抽象產品角色扮演。典型的應用就是 java.text.DateFormat 類別，一個抽象產品類別同時是子類別的工廠，如下圖所示。本章在下面給出了一個更爲詳盡的描述。

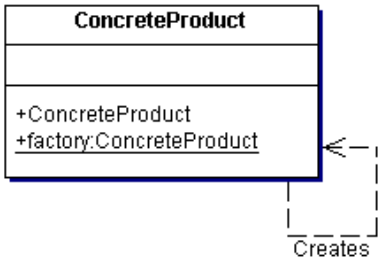


三個角色全部合併

如果抽象產品角色已經被省略，而工廠角色就可以與具體產品角色合併。換言之，一個產品類別爲自身的工廠，如下圖所示。



如果仍然以前面討論過的示意性系統爲例，這個系統的結構圖如下所示。



顯然，三個原本獨立的角色：工廠角色、抽象產品以及具體產品角色都已經合併成爲一個類別，這個類別自行建構自己的實例。請見下面的程式碼。

程式碼清單 13：具體產品角色與工廠角色合併後的程式碼

```
package com.javapatterns.simplefactory.simplified1;
public class ConcreteProduct
{
    public ConcreteProduct(){}
   /**
    *   靜態工廠方法
    */
    public static ConcreteProduct factory()
    {
        return new ConcreteProduct();
    }
}
```

這種退化的簡單工廠模式與單例模式以及多例模式有相似之處，但是並不等於單例或者多例模式。關於這一點的討論請參見本章後面的敍述。

產品物件的循環使用和登記式的工廠方法

由於簡單工廠模式是一個非常基本的設計模式，因此它會在較爲複雜的設計模式中出現。在本章前面所給出的示意性例子中，工廠方法總是簡單地呼叫產品類別的建構方法以建構一個新的產品實例，然後將這個實例提供給用戶端，而在實際情形中工廠方法所做的事情可以相當複雜。

在本書所討論的所有設計模式中，單例模式與多例模式是建立在簡單工廠模式的基礎之上的，而且它們都要求工廠方法具有特殊的邏輯，以便能循環使用產品的實例。

在很多情況下，產品物件可以循環使用。換言之，工廠方法可以循環使用已經建構出來的物件，而不是每一次都建構新的產品物件。工廠方法可以通過登記它所建構的產品物件來達到循環使用產品物件的目的。

如果工廠方法總是循環使用同一個產品物件，那麼這個工廠物件可以使用一個屬性來存儲這個產品物件。每一次用戶端呼叫工廠方法時，工廠方法總是提供這同一個物件。在單例模式中就是這樣，單例類別提供一個靜態工廠方法，向外界提供一個惟一的單例類別實例。

如果工廠方法永遠循環使用固定數目的一些產品物件，而且這些產品物件的數目並不大的話，可以使用一些私有屬性存儲對這些產品物件的引用。比如，一個永遠只提供一個產品物件的工廠物件可以使用一個靜態變數存儲對這個產品物件的引用。

相反，如果工廠方法使用數目不確定、或者數目較大的一些產品物件的話，使用屬性變數存儲對這些產品物件的引用就不方便。這時候，就應當使用聚集物件存儲對產品物件的引用。

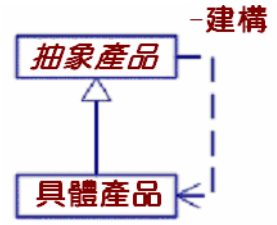
不管使用哪一種方法，工廠方法都可以做到循環使用它所建構的產品物件。循環的邏輯可能是基於這些產品類別的內部狀態，比如某一種狀態的產品物件只建構一個，讓所有需要處於這一狀態上的產品物件的用戶端共用這一個實例。

請參見下面對單例模式和多例模式的討論。

12.5　簡單工廠模式與其他模式的關係

單例模式

單例模式使用了簡單工廠模式。換言之，單例類別具有一個靜態工廠方法提供自身的實例。一個抽象產品類別同時是子類別的工廠，如下圖所示。



但是單例模式並不是簡單工廠模式的退化情形，單例模式要求單例類別的建構方法是私有的，從而用戶端不能直接將之實例化，而必須通過這個靜態工廠方法將之實例化，而且單例類別自身是自己的工廠角色。換言之，單例類別自己負責建構自身的實例。

單例類別使用一個靜態的屬性存儲自己的惟一實例，工廠方法永遠僅提供這一個實例。

多例模式

多例模式是對單例模式的推廣。多例模式與單例模式的共同之處在於它們都禁止外界直接將之實例化，同時通過靜態工廠方法向外界提供循環使用的自身的實例。它們的不同在於單例模式僅有一個實例，而多例模式則可以有多個實例。

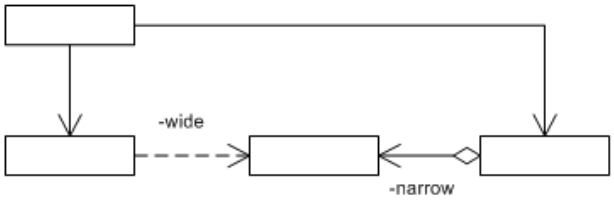
多例模式往往具有一個聚集屬性，通過向這個聚集屬性登記已經建構過的實例達到循環使用實例的目的。一般而言，一個典型的多例類別具有某種內部狀態，這個內部狀態可以用來區分各個實例；而對應於每一個內部狀態，都只有一個實例存在。

根據外界傳入的參量，工廠方法可以查詢自己的登記聚集，如果具有這個狀態的實例已經存在，就直接將這個實例提供給外界；反之，就首先建構一個新的滿足要求的實例，將之登記到聚集中，然後再提供給用戶端。

關於單例模式和多例模式請讀者參閱本書的“單例（Singleton）模式”一章和“多例（Multiton）模式”一章。

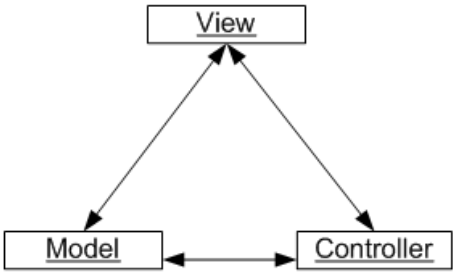
備忘錄模式

單例和多例模式使用了一個屬性或者聚集屬性來登記所建構的產品物件，以便可以通過查詢這個屬性或者聚集屬性找到和共用已經建構了的產品物件。這就是備忘錄模式的應用。備忘錄模式的簡略類別圖如下圖所示。

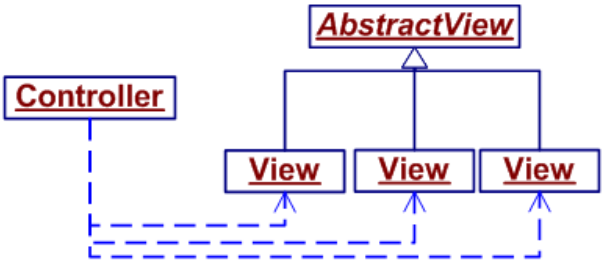


MVC 模式

MVC 模式並不是嚴格意義上的設計模式，而是在更高層次上的架構模式。MVC 模式可以分解成爲幾個設計模式的組合，包括合成模式、策略模式、觀察者模式，也有可能會包括裝飾模式、調停者模式、迭代子模式以及工廠方法模式等。MVC 模式的結構圖如下所是。關於 MVC 模式的討論可以參考本書的“專題：MVC 模式與用戶輸入資料檢查”一章。



簡單工廠模式所建構的物件往往屬於一個產品層級結構，這個層級結構可以是 MVC 模式中的視圖（View）；而工廠角色本身可以是控制器（Controller）。一個 MVC 模式可以有一個控制器和多個視圖，如下圖所示。



換言之，控制器端可以建構合適的視圖端，就如同工廠角色建構合適的物件角色一樣；而模型端則可以充當這個建構過程的用戶端。

如果系統需要有多個控制器參與這個過程的話，簡單工廠模式就不適用了，應當考慮使用工廠方法模式。請參閱本書的“工廠方法（Factory Method）模式”一章。

12.6　模式的優點和缺點

模式的優點

模式的核心是工廠類別。這個類別含有必要的判斷邏輯，可以決定在什麼時候建構哪一個產品類別的實例。而用戶端則可以免除直接建構產品物件的責任，而僅僅負責“消費”產品。簡單工廠模式通過這種做法實作了對責任的分割。

模式的缺點

正如同在本章前面所討論的，當產品類別有複雜的多層次層級結構時，工廠類別只有它自己。以不變應萬變，就是模式的缺點。

這個工廠類別集中了所有的產品建構邏輯，形成一個無所不知的全能類別，有人把這種類別叫做上帝類別（God Class）。如果這個全能類別代表的是農場的一個具體園丁的話，那麼這個園丁就需要對所有的產品負責，成了農場的關鍵人物，他什麼時候不能正常工作了，整個農場都要受到影響。

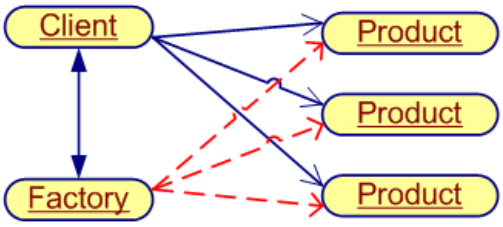
將這麼多的邏輯集中放在一個類別裏面的另外一個缺點是，當產品類別有不同的介面種類時，工廠類別需要判斷在什麼時候建構某種產品。這種對時機的判斷和對哪一種具體產品的判斷邏輯混合在一起，使得系統在將來進行功能擴展時較爲困難。這一缺點在工廠方法模式中得到克服。

由於簡單工廠模式使用靜態方法作爲工廠方法，而靜態方法無法由子類別繼承，因此，工廠角色無法形成基於繼承的層級結構。這一缺點會在工廠方法模式中得到克服。

“開–閉”原則

“開–閉”原則要求一個系統的設計能夠允許系統在無需修改的情況下，擴展其功能。那麼簡單工廠模式是否滿足這個條件？

要回答這個問題，首先需要將系統劃分成不同的子系統，再考慮功能擴展對於這些子系統的要求。一般而言，一個系統總可以劃分成爲產品的消費者角色（Client）、產品的工廠角色（Factory）以及產品角色（Product）三個子系統，如下圖所示。



在這個系統中，功能的擴展體現在引進新的產品上。“開–閉”原則要求系統允許當新的產品加入系統中，而無需對現有程式碼進行修改。這一點對於產品的消費角色是成立的，而對於工廠角色是不成立的。

對於產品消費角色來說，任何時候需要某種產品，只需向工廠角色請求即可。而工廠角色在接到請求後，會自行判斷建構和提供哪一個產品。所以，產品消費角色無需知道它得到的是哪一個產品；換言之，產品消費角色無需修改就可以接納新的產品。

對於工廠角色來說，增加新的產品是一個痛苦的過程。工廠角色必須知道每一種產品，如何建構它們，以及何時向用戶端提供它們。換言之，接納新的產品意味著修改這個工廠角色的程式碼。

綜合本節的討論，簡單工廠角色只在有限的程度上支持“開–閉”原則。

12.7　簡單工廠模式在 Java 中的應用

簡單工廠模式是一個很基本的設計模式，讀者可以在 Java 語言的 API 中看到這個模式的應用。下面就以幾個 Java API 中的應用爲例講解怎樣實作簡單工廠模式。

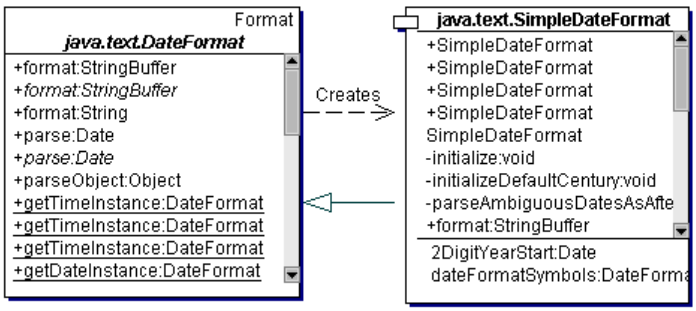
DateFormat 與簡單工廠模式

想必大多數的讀者都曾經使用工具類別 java.text.DateFormat 或其子類別來格式化一個本地日期或者時間，這個工具類別在處理英語和非英語的日期及時間格式上很有用處。

使用的目的

在本地機器上，時間和日期的格式存在一些標準的風格，比如一個日期可以被格式化爲 FULL、LONG、MEDIUM 或者 SHORT，但是它們所代表的意義隨著實作的不同而不同。

以英語爲例，SHORT 代表完全是數字型的短格式，例如“1/20/2002”或者“15:20”；而 MEDIUM 代表的中格式則是把縮寫文字增加到短格式中，比如：“Jul 22, 2002”或者“3:20pm”；而 LONG 代表的長格式用的是完整的詞，比如：“July 2, 2002”或者“3:20:10pm”；而 FULL 所代表的全格式包括了完整的日期資訊，比如：“Monday, July 22, 2002, AD”或者“3:20:10pm EST”等。DateFormat 與 SimpleDateFormat 的類別圖如下所示。



getDateInstance()方法

如果仔細考查這個 DateFormat 類別就會發現，它是一個抽象類別，但是卻提供了很多的靜態工廠方法，比如 getDateInstance()爲某種本地日期提供格式化，它由三個重載的方法組成。

程式碼清單 14：DateFormat 的部分程式碼

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style, Locale locale);
```

第一次接觸這個類別的初級程式師會有一些困惑，比如有的人會問，爲什麼一個抽象類別可以有自己的實例，並通過幾個方法提供自己的實例。實際上，一個抽象類別不能有自己的實例，這是絕對正確的。而應當注意的是，DateFormat 的工廠方法是靜態方法，並不是普通的方法。

getDateInstance()方法並沒有呼叫 DateFormat 的建構方法來提供自己的實例，作爲一個工廠方法，getDateInstance()方法做了一些有趣的事情。它所做的事情基本上可以分成兩部分：一是運用多型；二是使用靜態工廠方法。

從上面給出的 DateFormat 和 SimpleDateFormat 的類別圖可以看出，SimpleDateFormat 是抽象類別 DateFormat 的具體子類別，這就意味著 SimpleDateFormat 是一個 DateFormat 類型的子類型；而 getDateInstance()方法完全可以返回 SimpleDateFormat 的實例，而僅將它宣告爲 DateFormat 類型。這就是最純正的多型原則的運用[SINTES00]。

getDateInstance()方法是一個靜態方法。如果它是一個非靜態的普通方法會怎樣呢？要使用這個（非靜態）方法，用戶端必須首先取得這個類別的實例或者其子類別的實例。而這個類別是一個抽象類別，不可能有自己的實例，所以用戶端就只好首先取得其具體子類別的實例。如果用戶端能夠取得它的子類別的實例，那麼還需要這個工廠方法幹什麼呢？

顯然，這裏使用靜態工廠方法是爲了將具體子類別實例化的工作隱藏起來，從而用戶端不必考慮如何將具體子類別實例化，因爲抽象類別 DateFormat 會提供它的合適的具體子類別的實例。這是一個簡單工廠方法模式的絕佳應用。

針對抽象編程

這樣做是利用具體產品類別的超類別類型將它的真實類型隱藏起來，其好處是提供了系統的可擴展性。如果將來有新的具體子類別被加入到系統中來，那麼工廠類別可以將交給用戶端的物件換成新的子類別的實例，而對用戶端沒有任何影響。

這種將工廠方法的傳回類型設置成抽象產品類型的做法，叫做針對抽象編程，這是依賴倒轉原則（DIP）的應用。關於這一設計原則的詳細討論，請參見本書的“依賴倒轉原則（DIP）”一章。

本地時間

與本地日期的格式化相對應的是爲某種本地時間提供格式化，這一功能由三個重載的 getTimeInstance()方法提供。

程式碼清單 15：DateFormat 的部分程式碼

```
public final static DateFormat getTimeInstance();
public final static DateFormat getTimeInstance(int style);
public final static DateFormat getTimeInstance(int style, Locale locale);
```

顯然它們所提供的也是其具體子類別的實例，而不是自身的實例。因爲，它自身是一個抽象類別，不可能有自己的實例。由於其子類別必然是 DateFormat 的子類型，因此傳回類型可以是 DateFormat 類型，這也是多型的體現。

一個法語日期的例子

爲了進一步說明這個工具類別的使用辦法，在下面給出的例子中，假定本地語言是法語，並針對法語進行時間和日期的格式化。

程式碼清單 16：DateTester 的部分程式碼

```
package com.javapatterns.simplefactory.dateformat;
import java.util.Locale;
import java.util.Date;
import java.text.DateFormat;
import java.text.ParseException;
public class DateTester
{
    public static void main(String[] args)
    {
        Locale locale = Locale.FRENCH;
        Date date = new Date();
        String now = DateFormat.getTimeInstance(DateFormat.DEFAULT, locale)
            .format(date);
        System.out.println(now);
        try
        {
            date = DateFormat.getDateInstance(DateFormat.DEFAULT, locale)
                .parse("16 nov. 01");
            System.out.println(date);
        }
        catch(ParseException e)
        {
            System.out.println("Parsing exception:"+e);
        }
    }
}
```

其中 now 包含了按照法語格式寫出的當前時間，而 date 則讀入了以法語方式書寫的一個日期“16 nov. 01”。系統運行時會列印出下面的結果。

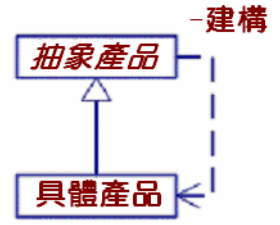
程式碼清單 17：DateTester 的運行結果

```
13:18:36
Fri Nov 16 00:00:00 PST 2001
```

簡單工廠模式的應用

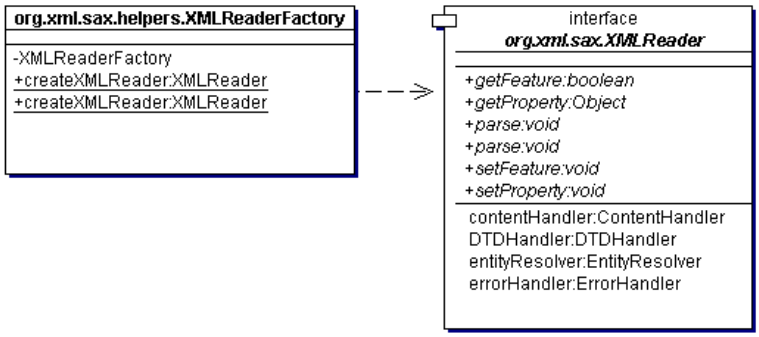
從上面的例子可以看出，由於使用了簡單工廠模式，用戶端完全不必操心工廠方法所傳回的物件是怎樣建構和構成的。工廠方法將實例化哪些物件以及如何實例化這些物件的細節隱藏起來，使得對這些物件的使用得到簡化。

與一般的簡單工廠模式不同的地方在於，這裏的工廠角色與抽象產品角色合併成一個類別。換言之，抽象產品角色負責具體產品角色的建構，這是簡單工廠模式的一個特例。如下圖所示。



SAX2 庫中的 XMLReaderFactory 與簡單工廠模式

在 SAX2 庫中，XMLReaderFactory 類別使用了簡單工廠模式，用來建構產品類別 XMLReader 的實例。下面是 XMLReaderFactory 和 XMLReader 類型的關係圖。



XMLReaderFactory 提供了兩種不同的靜態方法，適用於不同的驅動軟體參數。

關於 SAX2 庫的知識以及 SAX2 庫所涉及到的其他模式的討論，請閱讀本書“專題：XMLProperties 與適配器模式”和“專題：觀察者模式與 SAX2 流覽器”兩章。

12.8 女媧摶土造人

《風俗通》中說：“俗說天地開闢，未有人民。女媧摶黃土爲人。”女媧需要用土造出一個個的人，這就是簡單工廠模式的應用。

女媧摶土造人的思想便是簡單工廠模式的應用。現在本章就試圖使用 UML 和模式的語言來解釋女媧的做法。首先，在這個造人的思想裏面，有幾個重要的角色：女媧本身、抽象的人的概念和女媧所造出的具體的人們。

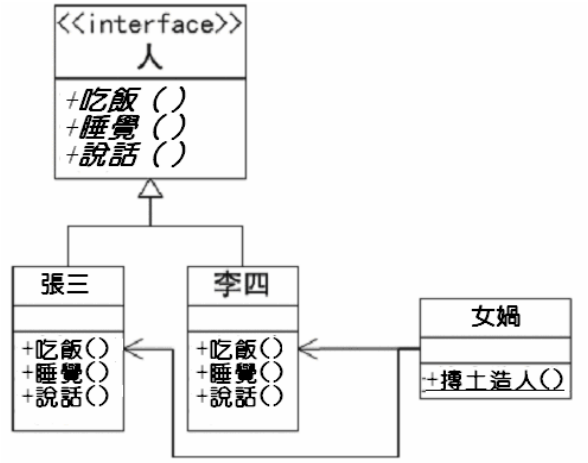
（1）女媧是一個工廠類別，也就是簡單工廠模式的核心角色；

（2）具體的一個個的人，包括張三、李四等。這些人便是簡單工廠模式裏面的具體產

品角色；

（3）抽象的人便是最早只存在於女媧的頭腦裏的一個想法，女媧按照這個想法造出的一個一個具體的人便都符合這個抽象的人的定義。換言之，這個抽象的想法規定了所有具體的人必須具有的介面。

女媧摶土造人的 UML 類別圖如下所示。



問答題

1. 在本節開始時不是說，工廠模式就是在不使用 new 操作符的情況下，將類別實例化的嗎，可爲什麼在具體實作時，仍然使用了 new 操作符呢?

2. 請使用簡單工廠模式設計一個建構不同幾何形狀，如圓形，方形和三角形實例的描圖員（Art Tracer）系統。每個幾何圖形都要有畫出 draw()和擦去 erase()兩個方法。 當描圖員接到指令，要求建構不支援的幾何圖形時，要提出 BadShapeException 例外。

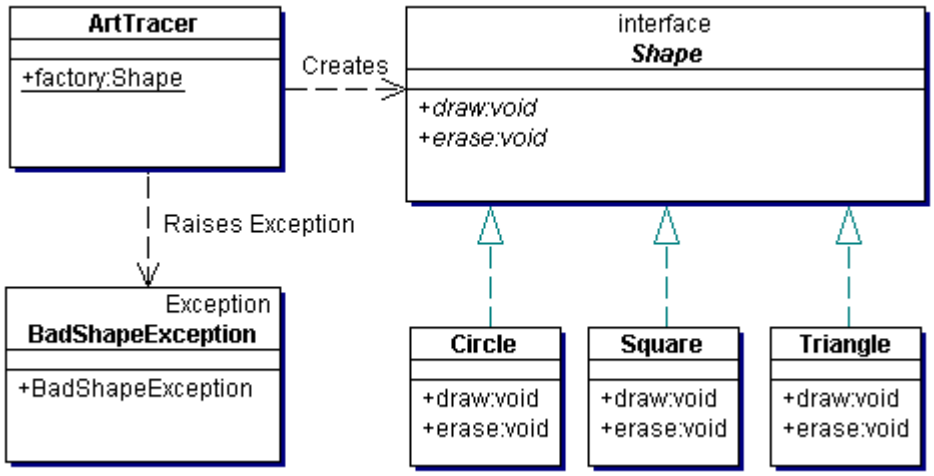
3. 請給出上一題的程式碼。

4. 請簡單舉例說明描圖員系統怎樣使用。

問答題答案

1. 對整個系統而言，工廠模式把具體使用 new 操作符的細節包裝和隱藏起來。當然只要程式是用 Java 語言寫的， Java 語言的特徵在細節裏一定會出現的。

2. 這裏給出問題的完整答案。描圖員（Art Tracer）系統的 UML 如下圖所示。



3. 描圖員系統包括一個介面扮演抽象產品角色，三個具體產品類別，一個工廠類別，和一個 Exception 類別。工廠類別便是 ArtTracer 類別，它的程式碼如下。

程式碼清單 18：ArtTracer 類別的程式碼

```
public class ArtTracer
{
    /**
     *  靜態工廠方法
     */
     public static Shape factory(String which) throws BadShapeException
     {
          if (which.equalsIgnoreCase("circle"))
          {
               return new Circle();
          }
          else if (which.equalsIgnoreCase("square"))
          {
               return new Square();
          }
          else if (which.equalsIgnoreCase("triangle"))
          {
               return new Triangle();
          }
          else
          {
               throw new BadShapeException(which);
          }
     }
}
```

Shape 是一個 Java 介面，規範出所有的產品類別必須實作的方法。它的程式碼如下。

程式碼清單 19：Shape 介面的程式碼

```
public interface Shape
{
    void draw();
    void erase();
}
```

Square 是一個具體產品類別，實作抽象產品角色，即 Shape 介面。

程式碼清單 20：Square 類別的程式碼

```
public class Square implements Shape
{
    public void draw()
    {
        System.out.println("Square.draw()");
    }
    public void erase()
    {
        System.out.println("Square.erase()");
    }
}
```

與上面的 Square 一樣，Circle 也是一個具體產品類別，實作 Shape 介面。

程式碼清單 21：Circle 類別的程式碼

```
public class Circle implements Shape
{
    public void draw()
    {
        System.out.println("Circle.draw()");
    }
    public void erase()
    {
        System.out.println("Circle.erase()");
    }
}
```

Triangle 是一個具體產品類別。與上面的兩個具體產品類別一樣，也實作了 Shape 介面。

程式碼清單 22：Triangle 類別的程式碼

```
public class Triangle implements Shape
{
    public void draw()
    {
        System.out.println("Triangle.draw()");
    }
    public void erase()
    {
        System.out.println("Triangle.erase()");
    }
}
```

爲了提供出錯管理，這裏特別提供一個 BadShapeException 例外類別。如果用戶端請求

一個並不存在的 Shape 的話，工廠就應當拋出這個例外。

程式碼清單 23：BadShapeException 類別的程式碼

```
public class BadShapeException extends Exception
{
    public BadShapeException(String msg)
    {
        super(msg);
    }
}
```

4．描圖員(Art Tracer)系統的使用方法如下。

程式碼清單 24：描圖員(Art Tracer)系統的程式碼

```
try
{
    ArtTracer art = new ArtTracer();
    art.factory("circle");
    art.factory("square");
    art.factory("triangle");
    art.factory("diamond");
}
catch(BadShapeException e)
{
    ...
}
```

對 ArtTracer 類別提出菱形（diamond）請求時，會收到 BadShapeException 例外。

參考文獻

[SINTES00] Tony Sintes, Polymorphism in its Purest Form - The Nature of Abstract Classes and Polymorphism, JavaWorld (www.javaworld.com), December 2000.